A High Performance Implementation of Prolog

Michael O. Newton

Computer Science Department
California Institute of Technology

5234:TR:86

A High Performance Implemcntation of Prolog

Michael O. Newton

Technical Report TR:5234:86

April 10, 1987

Caltech Computer Science Department

Pasadena, California 91125

Abstract

We discuss an efficient implementation of the Warren Abstract Machine (WAM) [12] in detail. Special attention is given to data formats, memory layout, WAM optimizations and code generation techniques. A final section describes some hardware considerations for even higher performance execution. Currently the compiler produces code that runs at approximately 900,000 logical inferences per second (LIPS) on a single processor of an IBM 3090 using the naive reverse benchmark. Using several of the yet unimplemented optimizations, we expect this figure to top one million LIPS.

1 Introduction

In 1977 David Warren published a set of papers on an efficient Prolog compiler for the DEC-20 series of computers [9, 10, 11]. The compiled code showed order of magnitude improvements over previous interpreters – executing at approximately 40,000 LIPS on a DEC-20. Later, in October 1983, Warren presented a new method of compiling Prolog [12], through the use of an intermediate code for a virtual machine usually referred to as the Warren Abstract Machine (WAM). Familiarity with the above references is required for understanding this paper.

In 1982, with support from IBM, Caltech decided to do an implementation of this abstract machine. The current implementation is nearly finished and consists of approximately 8000 lines of Prolog code (the compiler, optimizer, merger and WAM assembler) and 4000 lines of assembly (the run time system). By writing the compiler itself in Prolog, we were faced with many realities we would otherwise have ignored – size, debugging and garbage collection issues that are easily overlooked with trivial test cases. The compiler completely compiles all of itself.

This paper describes various implementation details for a high performance version of the Warren virtual machine. While a few techniques discussed will be specific to the IBM 370 series of computers, these will be in the minority.

The sections are arranged in a bottom up hierarchy. The first two sections describe the lowest level: data types and memory layout. The next section describes aspects of code generation. Following this are some details of the highest levels: a summary of various changes that were made to the WAM for performance reasons, and a brief overview of the large subroutines – the unifier, the fail code, and evaluatable predicates. The final section relates architectural considerations the author feels would have a major impact on the speed of compiled Prolog code. The appendices include WAM, optimized WAM and assembly code for a small sample program.

Throughout the paper references will be made to instructions and data formats described in [12]. The reader is expected to be familiar with this paper. In all samples, WAM code is shown as Prolog terms. In those (painful) cases when 370 assembly language must be shown, it will be without regard for 'card' columns.

1.1 Acknowledgements

This work would not have been done without the help and comments of Jim Kajiya and Keith Hughes. Working with both of these people has been very enjoyable and interesting. A good overview of our compiler is in [1]. Some of the ideas in this paper came as a result of long conversations with Ross Overbeek at Argonne National Labs (see also [4]).

1.2 Other Work

A few of the design decisions were the result of the useful statistics published by Evan Tick [6]. In addition the reader is referred to the paper by Andrew K. Turk [7], who independently reports some of the same optimizations, but who also has several "higher-level" optimizations.

2 Data Types

Prolog programs spend most of their time searching through clauses, looking for matches. This

'matching', called unification, is highly dependent on the types of data items involved. Thus, there is a great need for a fast way of determining the type of data pointed to by an object. Two possible method are in general use: separate areas for different data types, or tagged data items.

We have taken the tagged data item approach. While not ideal on the new, larger XA series of IBM's, for most System/370 architecture machines it is ideal – there is a 'wasted' high order byte in every pointer word that lends itself very nicely to tagging. By appropriately picking these tags, significant speedups (factors of 2-4) can be obtained.

There are only a few primitive types necessary in a Prolog Architecture:

- pointers – these can point at any data type, including themselves.
- number – integers, floating point and possibly others.
- atoms – 'string' constants.
- structures – composed of any data type.

All user data is built out of these data types, mainly through the use of structures. Thus, only a small number of distinct tags are needed to encode the various data types. However, in addition to these bits, a garbage collection bit is also necessary. Throughout this paper, "simple data type" refers to any data type except a structure and a pointer is said to point to an object if it points to anything but another pointer. Note that pointer chains do occur.

Integers, pointers, and atoms each consist of a tag and then the data, sometimes in the form of an address. The data format format of a structure is more implementation dependent. The representation will depend on the choice of tags and the available instructions as well as detail of how stacks grow and shrink.

2.1 Tags

Of the data format decisions, none is more important than the tags applied to the various data types. In the execution of Prolog only like terms are compared, so comparing tags is the single most frequent operation. These changes can easily produce a factor of two or three in performance. Note that most of the commments apply only to untagged machine architecures. Tagged architectures are preferable, but rarer.

2.1.1 Pointers

When inspecting tags in a WAM instruction the most common decision is whether an item is a pointer or not. Thus, in our implementation the sign bit of each word determines if the item points to an object or to another pointer. As will be shown during the discussion on dereferencing, this will have major impact on the speed of the compiler.

If the item is not a pointer, the next most common decision is between a structure or list and an atom, number, or constant. To determine this, we use the second highest order bit to indicate a structured data item. On the IBM architecture, this has a second advantage – the testing and branching can be done in one instruction (very important on 3090 type architectures where instruction count needs to be kept minimal) as shown here:

```
BXLE  R1,R1,TAGSET      % Branches if second highest order
                        % bit is set (structure).  R1 may
                        % be any odd numbered register. R1
                        % is destroyed by this operation.
```

If the item is a pointer, we must determine if it is a free variable or part of a pointer chain. There are several possible ways of representing this choice:

- As above, by using the second highest order bit.
- By having a free variable point to itself.
- By having a free variable be stored as a zero.

Which of these should be chosen is highly architecture dependent. I will discuss the differences in detail.

The first choice, using the second highest order bit is a simple but effective way of implementing unbound variable tagging. Usually however, one of the other two methods would be quicker, as bit tests are usually not as efficient as comparisons or sign tests.

The second possibility has the advantage that it is very robust – it is very unlikely that a random location in memory points to itself without being set up this way. This is very useful when debugging a compiler. However, it involves several inefficiencies. One of these is during garbage collection. Each free variable that is moved must have its address translated. In addition, to test for a free variable, one must do a memory reference, so a performance penalty is likely.

The final possibility, storing a free variable as a zero, is risky when first debugging a compiler – there are many other ways that a zero can be located in memory. But, the advantages are numerous. Instead of the need for doing memory references to determine whether a variable is free or not, only a simple test must be done. When storing multiple free variables (as in `allocate(N)`), one needs only do a block copy of zeroes.

2.1.2 Lists as structures

Another minor modification to the WAM in our implementation was the removal of the distinct tags for structures and list. This is most noticeable in the `switch_on_term` instruction, which now only takes two arguments – the place to go for atoms, and the place to go for structures. We did this so that unify code (in-line and in the unifier) would not have to detect whether a list coded with a tag was matching a list coded with a functor of period.

2.1.3 System/370 particulars

Unfortunately, on the System 370, there is no single instruction to store a zero. And, since the need to test for a free variable happens in less than 35 percent of dereferencing loops [6], it is slightly more efficient to have variables point to themselves (ignoring time spent garbage collecting). At one point we almost reserved a register to contain zero, but the supply of general purpose registers in this architecture is limited, and we believed a free register would have been better spent on other constants. Since then we have been seriously considering using a zero – see later.

In the (original) System/370 architecture, memory address are 24 bits wide, and the high order byte of each word lends itself nicely to tagging. In the later, extended addressing, architecture, up to 31

bits may be used for addressing, however, most of the 'virtual machines' still run in 24 bit address mode. While we implemented tagging in the high order byte of address words, it would be relatively simple to put the tags in separate words, and have full extended addressing. This would result in a slight decrease in speed due to a higher number of memory references. Current estimates would place the decrease around 30 percent – since the sign bit can still be used for tagging.

2.2 Terms

Prolog semantics imply a right to left, depth first order during unification of terms. To keep the number of memory accesses to a minimum, it is critical to lay out terms in as efficient manner as possible. On some machines small integer offsets work well in one direction but not in another. This would affect the direction of a term. As another example, to detect the end of a term, there are several choices – in some systems there will be enough bits in a word to include some flag bits, while in other systems a seperate word would be needed, and in others the arity could be stuck in a header word.

The layout of terms in memory is affected by several other decisions – the way bytes and words are stored in memory, the ease of detecting the end of a string of words and the design of the unifier.

2.2.1 Direction of terms in memory

After checking for similar functors and arities, unification in Prolog proceeds from left to right. Thus, when building a structure, the most efficient method of storing the data in the order:

- functor/arity,
- argument 1,
- argument 2,
- ...
- argument n.

Currently stacks grow downwards so terms are built within the heap (and code area) in this direction.

2.2.2 The end of a term

We considered two possible ways of determining the end of a term in memory. The first solution was to store a flag of zero or negative one at the end of the term. However, this had several disadvantages:

- When unifying different arity terms many unifies will be done before the unifier will notice that the arities are different.
- When unifying an extra memory reference must be done at the end of each structure.
- When garbage collecting, the garbage collector must determine the size of the structure to be moved and then do some calculations.
- It wastes memory.

Since the functor pointer in the first word of our data structure was just a address pointer into the atom table, there was an unused byte at the top of the structure. This was used to store the arity. Though limiting the arity of structures to 256 (actually 170 due to a garbage collection bit), this was considered a minor limitation compared with the above inefficiencies.

2.3 Other data items

In the initial implementation, integers and floating point numbers are stored as a tag and the number. We plan on expanding this, so that a full 32 (64 for floating point) bits can be used. Atoms are stored as a tag and a pointer to the character string, and references are stored as a tag and a pointer value. In each of the later cases, extended addressing could be accomplished by separating the tag and data into separate words – with only a minor performance penalty.

3 Memory Layout

The memory layout that we choose were very close to the recomendations in [12]. Those changes that were made either reflected either the fact that we were not using a byte coded interpreter, or the peculiarities of the architecture.

Overall, there are seven data areas:

- Compiled Code.
- Push Down List – for the unifier.
- Trail – tracks the variables that are bound for the failure code.
- Environments – that keep track of the state of the machine.
- Choice Points – that represent alternative not yet followed.
- Heap – the global memory area.
- Data space – for *asserts* and *retracts*.

Note that we have split Warren's general stack into an environment stack and a choice point stack so that we could save memory by removing unused choice points. In addition, [12] does not mention code space as no mention is given of *assert's* and *retract's*.

Of these areas, the PDL, Trail, Environments, Choice Points and Heap all act like stacks. Structures are stored only in the heap and code areas. In addition, a frequent needed piece of information is whether the structure being built is in the "most recent" area of the Heap.

3.1 Stacks

3.1.1 Direction

As already mentioned, a structure consists of a structure tag and a pointer to the structures elements. The first word pointed to at this location has a tag representing the arity of the function and a pointer

to the functor name (atom) location. The next (growing downwards on the IBM) arity number of words represent the subterms.

Since, for almost all the stacks, we wish to reference areas off the top of the stack, all stacks grow downward. This is only necessary on 370 style architectures, where base plus (positive) offset addressing is the only addressing mode. Should this be done, the order of the stacks, as suggested in ([5]) should be reversed, with the code area, and the heap as the top and next to top stacks in memory. As noted below, this permits efficient trailing.

Unfortunately – for a System 370 architecture – we decided to have the heap grow in the same direction as all the other stacks. This was a mistake: to access a structure in read mode we would subtract the structure pointer to the end of the structure and use positive offset; and to build an item in write mode, we would push a word at a time. If, instead, the code space and the heap were put in low core relative to the other stacks, building a structure could be done with cheaper instructions and reading a structure would take at most one subtract (as opposed to an always needed subtract). With this changed, we would expect speeds of well over a 1,000,000 LIPS on a single processor of a 3090. We hope to try this out in the near future.

3.1.2 Overflow detection

We do not explicitly test for stack overflow. Rather, we put a special page at the top and bottom of each stack. Using the storage key protection mechanism (virtual memory mapping) mechanism of the IBM 370 architecture, overflow can be detected by trapping thereby saving valuable cycles.

If this alternative is not available on a particular machine, another possibility is to have the compiler compute the maximum number of words each procedure can push onto each stack. Then, at the beginning of each procedure, a test can be done to see if stack overflow is possible.

3.1.3 Order of stacks for trailing

Trailing is only performed when binding a variable in the code space, or in the heap. However, when trailing the heap, this need only be done when binding a pointer in an older area of the heap than the current frame.

Another consideration in the ordering of stacks is trailing. By having the heap grow towards the code section, and away from the other stacks, a simple register comparison ($R \geq HB$) tells us if the new address is in a portion of the heap that must be trailed.

Note that we do not use the unsafe variable distinction of [12]. This is reflects our choice of free variable representation and the expense of checking variables. Since the WAM argument registers are located in memory and a free variable points to itself, the unsafe variable instructions would necessitate a test and then a modification of the register to a newly set up location in the heap if the variable was still unbound. Rephrased, a register pointing to itself is harder to handle than having all variables located in the heap.

When we change our representation of a free variable to a zero instead of a pointer that points to itself, then we will also restore the unsafe instructions.

3.1.4 Block moves of memory and registers

Finally, note that the ordering of the prolog virtual registers (A0...An in ([5])) is important. Often,

blocks of these are moved as a single unit, so it is helpful for the order to be the same as that saved by the `try me else` and similar instructions.

3.2 Globally Addressable Segment

In certain machines, global memory layout is very important. In particular, certain constants, save areas and routines are widely used. By making these easily addressable, memory fetches for the addresses of these values can be saved.

The problem is especially acute on BASE+OFFSET type architectures that do not provide in-line constants (IBM 370, 80x86). In our version of the compiler, we reserve a special register to always point to this "globally addressable" memory segment. This saved loading a pointer to such important constants as `NILPTR` (for checking ends of lists), the constant four (for memory addressing), and various tag values. That the unifier and fail code were also there meant that routines could branch straight to the code rather than having to load an address and then branch.

For a further increase in speed, the first item in this global area was the constant '[ ]'. Thus, we could use the low core pointer (appropriately tagged, and always residing in a register), in any WAM instructions that tested for nil or for the end of a list.

3.2.1 Often used constants

The prolog system starts with certain atoms predefined – '[ ]', ' ', 'user', and '.'. Note that `X = ' '` is the result of `name([],X)`.

By having the atoms stored in the form:

```
        DefConst    PtrTo(NextAtom - 4)
ATOM:   DefConst    Lenght(Atom),
        DefConst    C'Atom'
```

the atom insertion code is relatively fast – one only has to do a long string comparison on those atoms which have the same number of characters.

Another major decision was the Argument registers. It would be possible to put the low order two or three WAM argument registers into real registers and gain speed in the standard 'append/3' timings – which does very few non-deterministic procedure invocations. However, in any real Prolog program there would be many more choice points, and the constant need to store and retrieve WAM machine registers like `CP`, `HB`, and `H` would slow execution down. Ideally there should be enough real machine registers for both these registers and the argument registers, but this is not the case on any currently popular machine, so we put these constants into a globally addressable segment for efficient access.

3.2.2 Evaluatable predicates and common routines

Presently, in our implementation, most of the (non-IO) evaluatable predicates are also in the globally addressable segment. However, certain ones of these – var/1, integer/1, repeat, fail, true, atom/1 ... should be coded for inline by the code generator. This is a very easy optimization. Ideally though, the compiler front end would detect situations like:

```
foo(X) :- var(X), blech(X).
foo(X) :- integer(X), twilde(X).
foo(X) :- float(X), blorp(X).
ufoo([X|Y]) :- blechity(X), foo(Y).
```

and generate appropriate code – the switch_on_term instruction would automatically go to the right routine, and no choice points would be ever set up.

4 WAM Code generation

4.1 Registers

Several of our optimizations result in better register utilization. One of these depends on the interaction between switch_on_term and getstruct. Since switch_on_term must already dereference argument register zero, if the next get or put instruction references the same register, the value is already loaded.

When in read mode after a getstruct, the following unifies occur off of the S register. After each unify the S register is updated to the next item. However, it is possible to point to the last item in the term (given that getstruct knows the arity of the functor) and then do all memory references off of an offset. This can save several pointer subtractions on some machines.

In the flow control instructions, the argument register for the current procedure are saved and restored. To expedite this, we use block moves (STM and LM on a 370) and appropriately ordered the registers in memory and on the backtrack stack.

An interesting observation is that for normal flow through WAM instructions, only three work registers were used (above the normal WAM registers). The only places were more registers are needed are the unifier, the fail code and the evaluatable predicates.

4.2 Trailing

Trailing receives relatively little coverage in the Warren paper, in a real implementation, however, it is very costly – either one must trail every time one binds a variable (extremely expensive in terms of memory and memory speed), or one must test whether one needs to trail. If stacks are ideally ordered this test consists of comparing the HB register with the top of the Heap, and trailing if H < HB (ie: binding a variable in a non-local stack frame). If stacks are not ideally ordered, one must also test to make sure that the address is withing the heap. In either case, in most implementations a sequence like:

```
        CR      H,HB
        BLE     NoTrail
        ST      H,Trail
        S       H,FOUR
NoTrail:
```

must be coded. On a fast machine, the branch, which is often taken, destroys pipelines. In addition, on many machines, one wishes to branch to places that are ideally located (full word/double word

boundaries). In the above code, 'NoTrail' is reached through both branches, so appropriate alignment is difficult.

In reality, trailing usually need not be done. Consider the test case, append/3:

```
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
append([],L,L).
```

along with the mode declaration:

```
:- mode append(+,+,?).
```

We are guaranteed that the first argument, the input list is instantiated. Then, append takes the tail of this argument, and recursively calls itself. Thus, the first argument must be fully instantiated! This is easily detectable by flow analysis.

If the first argument is fully instantiated, there is no point in trailing the third argument in all of the recursive calls – if append ever fails in the middle of execution, it will fail all the way back to the beginning. Thus, we can change the WAM code for append/3 to never trail after the first call. (See the appendices).

To do this, we introduce several new WAM instructions. The first, `trail_it` causes the immediate trailing of an argument. `trailing(on)` and `trailing(off)` are meta-instructions telling the code generator when it need not generate trail code when producing code for instructions like `getstruct` or any of the unify instructions. The net effect is to save many branches with the large loss of performance suffered by a broken pipeline.

4.3 Dereferencing Loop

The dereferencing loop is the single most commonly executed piece of code in the system. It therefore is also one of the most time critical. Given hardware that will not dereference automatically, it is most important to arrange tagging and the representation of pointers so that it can be executed as quickly as possible. As Tick points out in [6], 66 percent of the time dereferencing terminates immediatedly with a bound item. Another 33 percent of the time only one loop is necessary. Based on this we try to test if an item is bound as soon as possible. In addition, we unfold the loop once, so that if the item is a pointer, a loop is only necessary after first dereferencing once.

At the same time, one of the times that dereferencing is most often done is in the instruction `switch_on_term`. As already mentioned, we keep the bound value around in the `S` register if the next instruction uses argument register zero. Also, in the `switch_on_term` deref loop, we need to decide if the atom is bound to an atom or to a structure. Since a structure is much more common, this occurs as a special first test in the deref loop.

4.4 Atoms

Prolog atoms and functor names are stored as strings in a special area. It is possible to detect equality of atoms by doing a string comparison, but it is much more efficient to do this only at the time of atom creation. New atoms can only be created by `read`ing or through the metalogical predicate `name/2`. When either of these encounters an atom, it calls the atom insertion code with

a pointer to the string length and the string itself. The atom creation code then checks the the linked list of atoms, searching for an equal. If none is found the new atom is inserted into the list. From then on, atom comparisons can be done solely by an address compare. Ideally the linked list of atoms would be a sorted binary tree, but we have not yet felt this was worth the coding time.

5 WAM Optimizations

5.1 Elimination of Read and Write Modes

One of the easiest ways of making the Warren Abstract Machine run faster is to eliminate read and write modes. These modes are set only by the two instructions **put_structure** and **get_structure**. Of these two instructions **put_structure** always leaves the machine in write mode, while the instruction **get_structure** leaves the machine in read or write depending on whether the argument is bound or not, respectively.

By generating the **unify** instructions that follow the **get_structure** or **put_structure** instruction directly into the code stream of the branch points, each of these **unify** instructions no longer needs to test for read or write mode. This saves time by eliminating branches and tests as well as preserving a relatively well sized pipeline.

5.2 Instruction Collapsing

Typically the instructions that follow a **get_structure** or **put_structure**, are very stereotypical. Usually, these instructions can be 'collapsed' together into a single more powerful WAM instruction that will generate less assembly language than its original constituents. One example of this is when several **gen_nil** instructions follow one another. By compiling these as a single WAM instruction several memory access to the nil pointer can be eliminated. Another example is an instruction that calls the unifier followed by a **proceed**. In this case the unifier can be given the return address that the **proceed** would have branched to.

5.3 Listfollow to build structures efficiently

Another common activity is the building of lists. If the list is built one element at a time, from the head first, one can collapse the **unify** of the previous **put_structure**, the **put_structure**, and **unify** of the next value into one instruction. For cdr-coded machines, this can be done directly.

6 Prepackaged Routines

6.1 Unifier

The unifier is not called as often as we initially expected. This is largely due to the way that structures are built and taken apart in the clauses – all the work is explicitly stated and, thus, coded as WAM instructions. However, when the unifier is called, it has to do a fair amount of work.

Most of the optimizations already described in this paper, especially with regards to dereferencing and structure representation, not only speed the in line WAM code, but the unifier as well. If care

is taken when writing the unifier to handle the cases where one of the input arguments dereferences to a variable, a large percentage of calls will be handled in the most efficient manner. By having the unifier easily addressable, less time is spent on overhead to reach it.

For those cases where the two inputs are both complex structures, it is advantageous (at least on a System 370 architecture) to encode the unifier as a loop, using a small push down stack to store the triple:

- Pointer to where in term one the unifier is.
- Pointer to where in term two the unifier is.
- Number of arguments left at this level.

6.2 Fail code

The fail code is frequently used, and usually involves considerably more computation than unifying. Untrailing often is done on more trail items than is necessary, but with current machines, testing to avoid this work would take more time than the work itself.

6.3 Evaluatable Predicates

The evaluatable predicates are relatively straightforward to write, once one has decided the various data formats. The hardest part is detecting all the boundary conditions, like `name('',X)` or `name(X,[])`. For speed purposes, certain predicates like `var(X)`, `integer(X)`, or `atomic(X)` should be coded in line, or, if possible, as part of the `switch_on_term` instruction (see Chapter 3).

7 Future directions

Using the optimizations already implemented, we achieve speeds on the order of 870,000 LIPS. With the addition of the other optimizations we expect speeds on the order of 1,100,000 LIPS. The author believes that without major changes in hardware, Prolog machines could be easily built that would obtain roughly 5-10 million LIPS. To substantiate this claim, I will delve some into our particular code generator.

In the analysis of instruction timings on the IBM 3090, the overall rule was to keep instruction count to the minimum. Currently our main append loop consists of 30 instructions executing at the rate of 870,000 LIPS (Using optimizations already mentioned in this paper, but not yet implemented, will lower the instruction count to 28). Using the following optimizations, this would reduce to under 24 instructions. At the same time 4 memory references would disappear. These optimizations include:

- Base plus offset addressing considerations take away several cycles.
- Most of the remaining memory references can be placed in registers, these are:
  1. The argument registers.
  2. The list functor (./2).
  3. The structure tag.
  4. The constant four (for addressing).

- We have not yet implemented free variables as zeros, and,
- two instructions (BXLE and ICM) are used only for a fraction of what they do.

Simply having 32 registers instead of 16 would alleviate almost all of these inefficiencies.

The Prolog compiler only uses approximately ten percent of the instruction set of the IBM 3090. And, almost all branch targets and offset are small (less than 128) positive integers – thus the instruction set could be based on a very compact instruction set. In addition, certain key sequences (ST X,... , S X,=F'4') could be considered as individual instructions (push(X)). Combining these into single op codes would reduce memory size for the program and increase the effective instruction fetch/execution rate.

Ideally the WAM would be rebuilt at a lower level. The current code generator reflects this – most of the WAM instructions (like **get_structure** or **unify_value**) are built from a small set of primitives that map into one to eight IBM 370 instructions. The WAM is at too high a level – too many possible optimizations are lost, and 370 assembly is not the ideal target.

In addition to factors of two to four speed up in execution from the above changes, another possibility for increased performance is parallel execution. Though the Argonne group has obtained extremely large speedups on certain problems through the uses of independent execution on separate machines, another more fundamental approach would be applicable to almost all programs – two to four tightly coupled machines. These machines would share registers and data paths, and would execute different parts of the same procedure call. For example, in the **append** example used in the appendices, the two **get_list** instructions and their associated **unifies** could be done almost completely independently. Note also that the code for trailing and restoring from the trail can be done much more efficiently using a special purpose stack – only those locations that are still useful need be restored.

In the extreme, these tightly coupled machines could be considered as one machine with a very long instruction word (horizontal microcode). Each argument register would have associated with it a path to memory, a dereferencing engine and a very simple ALU.

A A sample program

All of the other appendices show WAM code related to the following sample program:

```
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
append([],L,L).

rev([H|T],Out) :- rev(T,TR), append(TR,[H],Out).
rev([],[]).
```

B Compiler WAM Code

The following is the output of our compiler immediately after code generation. No optimizations of any sort have been applied.

```
comment(procedure(/(append,3))).
```

```
'$LABEL'('L1').
switchonterm('$LABEL'('L4'),'$LABEL'('L5')).
try_me_else('$LABEL'('L6'),3).

'$LABEL'('L5').
getlist(a(0)).
unifyvar(t(0)).
unifyvar(t(1)).
getvar(t(2),a(1)).
getlist(a(2)).
unifyvalue(t(0)).
unifyvar(t(3)).
putvalue(a(0),t(1)).
putvalue(a(1),t(2)).
putvalue(a(2),t(3)).
execute('$LABEL'('L1')).

'$LABEL'('L6').
trust_me_else_fail.
'$LABEL'('L4').
getnil(a(0)).
getvar(t(0),a(1)).
getvalue(t(0),a(2)).
proceed.

comment(procedure(/(rev,2))).
'$LABEL'('L2').
switchonterm('$LABEL'('L7'),'$LABEL'('L8')).
try_me_else('$LABEL'('L9'),2).

'$LABEL'('L8').
allocate2(3,0).
getlist(a(0)).
unifyvar(p(1)).
unifyvar(t(0)).
getvar(p(2),a(1)).
putvalue(a(0),t(0)).
putvar(a(1),p(0)).
call('$LABEL'('L2')).
putvalue(a(0),p(0)).
putlist(a(1)).
buildvalue(p(1)).
buildnil.
buildend.
putvalue(a(2),p(2)).
deallocate.
execute('$LABEL'('L1')).

'$LABEL'('L9').
trust_me_else_fail.
'$LABEL'('L7').
getnil(a(0)).
getnil(a(1)).
proceed.
```

C Register Optimized WAM Code

After generating code, it is possible to do WAM register optimizations. We have not yet implemented this, but plan on doing so soon. The interested reader is referred to [8] and [7] for more details.

Notice that our compiler has produced extra labels (branch points). These are insignificant, as they do not produce any executable code.

```
comment(procedure(/(append,3))).
'$LABEL'('L1').
switchonterm('$LABEL'('L4'),'$LABEL'('L5')).
try_me_else('$LABEL'('L6'),3).

'$LABEL'('L5').
getlist(a(0)).
unifyvar(a(3)).
unifyvar(a(0)).
getlist(a(2)).
unifyvalue(a(3)).
unifyvar(a(2)).
execute('$LABEL'('L1')).

'$LABEL'('L6').
trust_me_else_fail.
'$LABEL'('L4').
getnil(a(0)).
getvalue(a(1),a(2)).
proceed.


comment(procedure(/(rev,2))).
'$LABEL'('L2').
switchonterm('$LABEL'('L7'),'$LABEL'('L8')).
try_me_else('$LABEL'('L9'),2).

'$LABEL'('L8').
allocate2(3,0).
getlist(a(0)).
unifyvar(p(1)).
unifyvar(a(0)).
getvar(p(2),a(1)).
putvar(a(1),p(0)).
call('$LABEL'('L2')).
putvalue(a(0),p(0)).
putlist(a(1)).
buildvalue(p(1)).
buildnil.
buildend.
putvalue(a(2),p(2)).
```

```
deallocate.
execute('$LABEL'('L1')).

'$LABEL'('L9').
trust_me_else_fail.
'$LABEL'('L7').
getnil(a(0)).
getnil(a(1)).
proceed.
```

D Optimized WAM Code

Either with or without register allocation, the WAM code is then run through our optimizer. This pass detects many of the optimizations cited in this paper. Here is a sample of the output:

```
comment(procedure(/(append,3))).
'$LABEL'('L1').
switchonterm(q('$LABEL'('L4')),q('$LABEL'('RL5'))).
try_me_else('$LABEL'('L6'),3).

'$LABEL'(fast('L5')).
getlist('SPHack',[unifyvar(a(3)),unifyvar(a(0))],'L5').
getlist(a(2),[unifyvalue(a(3)),unifyvar(a(2)),execute('$LABEL'('L1'))]).

'$LABEL'(fast('L6')).
trust_me_else_fail.

'$LABEL'('L4').
getnil(a(0)).
getvalue_proceed(a(1),a(2)).


comment(procedure(/(rev,2))).
'$LABEL'('L2').
switchonterm('$LABEL'('L7'),q('$LABEL'('L8'))).
try_me_else('$LABEL'('L9'),2).

'$LABEL'('L8').
allocate2(3,0).
getlist(a(0),[unifyvar(p(1)),unifyvar(a(0))]).
getvar(p(2),a(1)).
putvar(a(1),p(0)).
call('$LABEL'('L2')).
putvalue(a(0),p(0)).
putlist(a(1)).
buildvalue(p(1)).
buildnil.
buildend.
putvalue(a(2),p(2)).
```

```
deallocate.
execute('$LABEL'('L1')).

'$LABEL'(fast('L9')).
trust_me_else_fail.

'$LABEL'('L7').
getnils_proceed([a(0),a(1)]).
```

E Optimized WAM Code with Trailing Optimization

Until this point all code would run equally well with any variable as input or output, and no mode declarations have been used. However, by flow analysis of the program, the optimization discussed in the section on trailing can be applied. This results in the elimination of trailing code in the `getlist` and unify instructions. In addition, much of the flow control previously included (`try_me_else`, `trust_me_else_fail`) can be eliminated. (Unfortunately, this has no affect on speed.)

Included in the code are comments that explain most of the instructions that were added to the WAM instruction set.

Here is what the new code would look like:

```
comment(procedure(/(append,3))).
trail_it(a(2)).
trailing(off).

comment(internal_procedure(/(internal_append,3))).
'$LABEL'('L1').
switchonterm(q('$LABEL'('L4')),q('$LABEL'('RL5'))).

'$LABEL'(fast('L5')).
%%% 'SPhack' implies the argument is already in register 1, and
%%% has been dereferenced, all courtesy of 'switchonterm'.
%%%
%%% The following shows how unifies are incorporated
%%% into the respective get instructions.
getlist('SPHack',[unifyvar(a(3)),unifyvar(a(0))],'L5').
getlist(a(2),[unifyvalue(a(3)),unifyvar(a(2)),execute('$LABEL'('L1'))]).

'$LABEL'('L4').
getnil(a(0)).
%%% For effiency, we combine many pairs of: instruction, proceed
%%% as a single WAM instruction.  This increases prevents branches
%%% to branches.
getvalue_proceed(a(1),a(2)).

trailing(on).


comment(procedure(/(rev,2))).
```

```
'$LABEL'('L2').
switchonterm('$LABEL'('L7'),q('$LABEL'('L8'))).
try_me_else('$LABEL'('L9'),2).

'$LABEL'('L8').
allocate2(3,0).
getlist(a(0),[unifyvar(p(1)),unifyvar(a(0))]).
getvar(p(2),a(1)).
putvar(a(1),p(0)).
call('$LABEL'('L2')).
putvalue(a(0),p(0)).
putlist(a(1)).
buildvalue(p(1)).
buildnil.
buildend.
putvalue(a(2),p(2)).
deallocate.
execute('$LABEL'('L1')).

'$LABEL'(fast('L9')).
trust_me_else_fail.

'$LABEL'('L7').
getnils_proceed([a(0),a(1)]).
```

F IBM-370 Assembly

After the mandatory optimization pass, IBM 370 Assembly is produced. The code in this section is optimized for the 3090 series processors as much as our scant amount of data has allowed. In addition, markings are provided to indicate what code is executed in a typical loop through the two procedures. This is the code that currently executes at approximately 870,000 LIPS. Note that the trailing optimization is not included.

(A few of these optimizations have been done by hand, as mentioned earlier. The code, however, has run on a IBM 3090.)

```
* These equates represent the offset into each of the stacks
* for various values:
B$B      EQU     0
B$HB     EQU     4
B$TR     EQU     8
B$PLACE  EQU     12
B$CP     EQU     16
B$E      EQU     20
B$TOPE   EQU     20
B$CE     EQU     24
B$N      EQU     28
B$ARGS   EQU     32
B$A1     EQU     32
CE$CP    EQU     0
CE$CE    EQU     4
CE$CUT   EQU     8
CE$N     EQU     12
```

```
    CE$VARS  EQU      16
    * These are the various register assignments.  Many of the assignments
    * must be kept in this order.
    LC       EQU      R4                    * low core pointer
    BASE     EQU      R5                    * current addressibility
    S        EQU      R6                    * structure pointer
    H        EQU      R7                    * top of heap
    B        EQU      R8                    * top of backtrack point stack
    HB       EQU      R9                    * Heap backtrack
    TR       EQU      R10                   * top of trail
    URET     EQU      R11                   * return for the unifier
    CP       EQU      R12                   * continuation pointer
    E        EQU      R13                   * top of environment stack
    TOPE     EQU      R13                   * top of environment stack
    CE       EQU      R14                   * Current enviorement
    FLAGS    EQU      R14                   * system flags for tests
    *
    *
    *
    * --> comment(procedure(/(append,3)))
    * -----------> L1
--  L1       EQU      *
->           BALR     BASE,0
--           USING    *,BASE
    * --> switchonterm(q($LABEL(L4)),q($LABEL(RL5)))
->           L        R1,ARGS+4*0           * Get an arg-reg ptr
->  H1       LTR      S,R1                  * variable?
->           BNM      H2                    * No -- we're done
->           BXLE     R1,R1,RL5             * Sneaky !
             B        L4                    * quick branch
             DS       0D
    H2       C        R1,0(,R1)             * same ?
             BE       H0                    * yes -- we're done
             L        R1,0(,R1)             * otherwise go another level
             LTR      S,R1                  * variable?
             BNM      H2                    * No -- we're done
             BXLE     R1,R1,RL5             * Sneaky !
             B        L4                    * quick branch
             DS       0D
             LTORG    Here's a nice place
             DS       0D
    H0       EQU      *
    * --> try_me_else($LABEL(L6),3)
             LR       R3,B                  * prevback
    *        push(b) An...A1
             S        B,=F'44'              * save space on stack
             MVC      32(12,B),ARGS         * save it all
    *        push(b) N:
             LA       R15,3                 * save N
             L        URET,=A(L6)           * get where to go
    *        Push (b)  HB, TR, Place, CP, TOPE, CE & N:
             STM      HB,R15,4(B)           * efficently store!!
             ST       R3,0(,B)              * push(b) prevback:
             ST       R3,CUT                * CUT = prevback
             LR       HB,H                  * save H for backtracking
    * -----------> L5
    L5     EQU      *
    * --> getlist(SPHack,[unifyvar(a(3)),unifyvar(..)],..)
             LTR      S,S                   * R1 set by switchonterm
             BM       H3                    * go to right routine
    *        write mode----------------------
             O        H,STTTAG              * set tag bits
             ST       H,0(,S)               * set arg
             MVC      0(4,H),CDOTPTR        * get & store functor pointer
             SL       H,STTTAG4             * bump heap and clear tag
             CR       S,HB                  * Lower than HB?
             BL     H7                      * Yes, no need
    H6       ST       S,0(,TR)              * save it
```

```
          S       TR,FOUR                 * push it on the stack
    H7    EQU     *
    *     Build instructions ------------
    * unifyvar(a(3))
          ST      H,ARGS+4*3              * store value
          ST      H,0(,H)                 * and save new free var
          S       H,FOUR                  * and bump H
    * unifyvar(a(0))
          ST      H,ARGS+4*0              * store value
          ST      H,0(,H)                 * and save new free var
          S       H,FOUR                  * and bump H
          B       H4
          DS      0D
          LTORG   Here's a nice place
          DS      0D
    H3    EQU     *
--  RL5   EQU     *                       * Label for switchonterm
    *     Read mode----------------------
->        CLC     0(4,S),CDOTPTR          * see if same functor as heap
->        BNE     FAIL                    * No, go fail, else...
    *     clear struct tag, sub 4, and all other subs
--  H5    EQU      X'D0000000'+12         * compute
->        SL      S,=A(H5)                * do it
    *     Read instructions ------------
    * unifyvar(a(3))
->        MVC     ARGS+4*3(4),8(S)        * move what S points to
    * unifyvar(a(0))
->        MVC     ARGS+4*0(4),4(S)        * move what S points to
    *     End of getlist/getstruct------
    H4    EQU     *
    * --> getlist(a(2),[unifyvalue(a(3)),unifyvar(..),..])
->        ICM     S,X'F',ARGS+4*2         * Get an arg-reg ptr
->  H12   BM      H9                      * No -- we are done
->        C       S,0(,S)                 * same ?
->        BE      H13                     * yes -- we are done
          ICM     S,X'F',0(S)             * else, go another level
          BM      H9                      * No -- we are done
          C       S,0(,S)                 * same ?
          BE      H13                     * yes -- we are done
          B       H12
          DS      0D
--  H13   EQU     *
    *     write mode---------------------
->        O       H,STTTAG                * set tag bits
->        ST      H,0(,S)                 * set arg
->        MVC     0(4,H),CDOTPTR          * get & store functor pointer
->        SL      H,STTTAG4               * bump heap and clear tag
->        CR      S,HB                    * Lower than HB?
->        BL      H15                     * Yes, no need
->  H14   ST      S,0(,TR)                * save it
->        S       TR,FOUR                 * push it on the stack
--  H15   EQU     *
    *     Build instructions ------------
    * unifyvalue(a(3))
->        ICM     R1,X'F',ARGS+4*3        * Get an arg-reg ptr
->  H17   BM      H18                     * No -- we are done
          C       R1,0(,R1)               * same ?
          BE      H18                     * yes -- we are done
          ICM     R1,X'F',0(R1)           * else, go another level
          BM      H18                     * No -- we are done
          C       R1,0(,R1)               * same ?
          BE      H18                     * yes -- we are done
          B       H17
          DS      0D
--  H18   EQU     *
->        ST      R1,0(,H)                * and push value
->        S       H,FOUR                  * bump heap
    * unifyvar(a(2))
```

```
->          ST      H,ARGS+4*2              * store value
->          ST      H,0(,H)                 * and save new free var
->          S       H,FOUR                  * and bump H
    * execute($LABEL(L1))
->          BR      BASE                    * branch to where we should go
            DS      0D
            LTORG   Here's a nice place
            DS      0D
    H9      EQU     *
    *       Read mode----------------------
            L       R2,CDOTPTR
            CL      S,STTTAG                * Is it a struct?  STTTAG must
            BL      FAIL                    * be the largest (logical) tag
            C       R2,0(,S)                * see if same functor as heap
            BNE     FAIL                    * No, go fail, else...
    *       clear struct tag, sub 4, and all other subs
    H11     EQU      X'D0000000'+16         * compute
            SL      S,=A(H11)               * do it
    *       Read instructions ------------
    * unifyvalue(a(3))
            L       R1,ARGS+4*3             * Get an arg-reg ptr
            L       R2,12(,S)               * & adr of str drf 1
            BAL     URET,UNIFY              * call the unifier
    * unifyvar(a(2))
            MVC     ARGS+4*2(4),8(S)        * move what S points to
    * execute($LABEL(L1))
            L       BASE,=A(L1)             * load go to point
            BR      BASE                    * branch to where we should go
            DS      0D
            LTORG   Here's a nice place
            DS      0D
    *       End of getlist/getstruct------
    H10     EQU     *
    * -----------> L6
    L6      EQU     *
    * --> trust_me_else_fail
            LM      B,HB,B$B(B)             * Del. bcktrck pt & get old HB
            ST      B,CUT                   * CUT = B
    * -----------> L4
    L4      EQU     *
            BALR    BASE,0
            USING   *,BASE
    * --> getnil(a(0))
            ICM     R1,X'F',ARGS+4*0        * Get an arg-reg ptr
    H21     BM      H19                     * No -- we are done
            C       R1,0(,R1)                * same ?
            BE      H22                     * yes -- we are done
            ICM     R1,X'F',0(R1)           * else, go another level
            BM      H19                     * No -- we are done
            C       R1,0(,R1)                * same ?
            BE      H22                     * yes -- we are done
            B       H21
            DS      0D
    H22     EQU     *
            MVC     0(4,R1),NILPTR
            CR      R1,HB                   * Lower than HB?
            BL      H20                     * Yes, no need
    H23     ST      R1,0(,TR)               * save it
            S       TR,FOUR                 * push it on the stack
            B       H20
            DS      0D
            LTORG   Here's a nice place
            DS      0D
    H19     C       R1,NILPTR               *   = ?
            BNE     FAIL
    H20     EQU     *
    * --> getvalue_proceed(a(1),a(2))
            LM      R1,R2,ARGS+4*1
```

```
            LR      URET,CP
    *               put the CP in URET for the unifier
            B       UNIFY                   * and call the unifier
            DS      OD
            LTORG   Here's a nice place
            DS      OD
    *
    * --> comment(procedure(/(rev,2)))
    * . . .procedure(rev/2).....................
    * -----------> L2
--  L2      EQU     *
->          BALR    BASE,0
--          USING   *,BASE
    * --> switchonterm($LABEL(L7),q($LABEL(L8)))
->          L       R1,ARGS+4*0             * Get an arg-reg ptr
->  H27     LTR     S,R1                    * variable?
->          BNM     H28                     * No -- we are done
->          BXLE    R1,R1,L8                * Sneaky !
            L       R2,=A(L7)               * Go here? -- later optimize
            BR      R2                      * see code gnrtr for details
            DS      OD
    H28     C       R1,0(,R1)               * same ?
            BE      H26                     * yes -- we are done
            L       R1,0(,R1)               * otherwise go another level
            LTR     S,R1                    * variable?
            BNM     H28                     * No -- we are done
            BXLE    R1,R1,L8                * Sneaky !
            L       R2,=A(L7)               * Go here? -- later optimize
            BR      R2                      * see code gnrtr for details
            DS      OD
            LTORG   Here's a nice place
            DS      OD
    H26     EQU     *
    * --> try_me_else($LABEL(L9),2)
            LR      R3,B                    * prevback
    *       push(b) An...A1
            S       B,=F'40'                    * save space on stack
            MVC     32(8,B),ARGS                  * save it all
    *       push(b) N:
            LA      R15,2                   * save N
            L       URET,=A(L9)             * get where to go
    *       Push (b)  HB, TR, Place, CP, TOPE, CE & N:
            STM     HB,R15,4(B)             * efficently store!!
            ST      R3,0(,B)                * push(b) prevback:
            ST      R3,CUT                  * CUT = prevback
            LR      HB,H                    * save H for backtracking
    * -----------> L8
--  L8      EQU     *
->          BALR    BASE,0
--          USING   *,BASE
    * --> allocate2(3,0)
->          LR      URET,TOPE               * save TOPE
->          L       R15,CUT                 * get cut
    *       Push N1 new vars:
->          S       E,FOUR                  * inc pointer
->          ST      E,0(,E)                 * make a free var
->          S       E,FOUR                  * inc pointer
->          ST      E,0(,E)                 * make a free var
->          S       E,FOUR                  * inc pointer
->          ST      E,0(,E)                 * make a free var
->          LA      R0,3                    * Get N1 in a reg
->          S       TOPE,=F'24'             * Do the change to TOPE
    *       push CE, cut, N1:
->          STM     CE,0,12(E)              * push*2
->          STM     URET,CP,4(E)            * push old TOPE:, CP:
->          LA      CE,8(,E)                * set current to top
    *       push N2:
->          LA      R2,0                    * Get N2 in a reg
```

```
->          ST      R2,0(,E)              *   and push it
->          LR      R1,E                  * Get E - 1
->          S       R1,FOUR
->          LA      R0,0                  * Get N1 in a reg
->          ST      R0,0(,R1)             * Store N2
    *       push n2 new vars
    * --> getlist(a(0),[unifyvar(p(1)),unifyvar(a(..))])
->          ICM     S,X'F',ARGS+4*0       * Get an arg-reg ptr
->  H32     BM      H29                   * No -- we are done
            C       S,0(,S)               * same ?
            BE      H33                   * yes -- we are done
            ICM     S,X'F',0(S)           * else, go another level
            BM      H29                   * No -- we are done
            C       S,0(,S)               * same ?
            BE      H33                   * yes -- we are done
            B       H32
            DS      0D
    H33   EQU     *
    *       write mode---------------------
            0       H,STTTAG              * set tag bits
            ST      H,0(,S)               * set arg
            MVC     0(4,H),CDOTPTR        * get & store functor pointer
            SL      H,STTTAG4             * bump heap and clear tag
            CR      S,HB                  * Lower than HB?
            BL      H35                   * Yes, no need
    H34     ST      S,0(,TR)              * save it
            S       TR,FOUR               * push it on the stack
    H35     EQU     *
    *       Build instructions ------------
    * unifyvar(p(1))
            ST      H,CE$VARS+4*1(,CE)    * store
            ST      H,0(,H)               * and save new free var
            S       H,FOUR                * and bump H
    * unifyvar(a(0))
            ST      H,ARGS+4*0            * store value
            ST      H,0(,H)               * and save new free var
            S       H,FOUR                * and bump H
            B       H30
            DS      0D
            LTORG   Here's a nice place
            DS      0D
--  H29     EQU     *
    *       Read mode----------------------
->          L       R2,CDOTPTR
->          CL      S,STTTAG              * Is it a struct?  STTTAG must
->          BL      FAIL                  * be the largest (logical) tag
->          C       R2,0(,S)              * see if same functor as heap
->          BNE     FAIL                  * No, go fail, else...
    *       clear struct tag, sub 4, and all other subs
--  H31   EQU     X'D0000000'+12          * compute
->          SL      S,=A(H31)             * do it
    *       Read instructions ------------
    * unifyvar(p(1))
->          MVC     CE$VARS+4*1(4,CE),8(S) * move what S points to
    * unifyvar(a(0))
->          MVC     ARGS+4*0(4),4(S)      * move what S points to
    *       End of getlist/getstruct------
--  H30     EQU     *
    * --> getvar(p(2),a(1))
->          ICM     R1,X'F',ARGS+4*1      * Get an arg-reg ptr
->  H37     BM      H38                    * No -- we are done
            C       R1,0(,R1)              * same ?
            BE      H38                   * yes -- we are done
            ICM     R1,X'F',0(R1)         * else, go another level
            BM      H38                   * No -- we are done
            C       R1,0(,R1)              * same ?
            BE      H38                   * yes -- we are done
            B       H37
```

```
            DS      0D
--  H38     EQU     *
->          ST      R1,CE$VARS+4*2(,CE)   * store
    * --> putvar(a(1),p(0))
->          ST      H,ARGS+4*1            * store value
->          ST      H,CE$VARS+4*0(,CE)    * store
->          ST      H,0(,H)
    *               and push free variable onto heap
->          S       H,FOUR                * and bump heap
    * --> call($LABEL(L2))
->          L       CP,=A(H39)            * and return point
->          BR      BASE                  * and go to routine
            DS      0D
            LTORG   Here's a nice place
            DS      0D
            USING   *,BASE                * Re-establish addresibility
    H39     LR      BASE,CP               * a quicker way
    * --> putvalue(a(0),p(0))
            MVC     ARGS+4*0(4),CE$VARS+4*0(CE)       * Do a MVC for speed
    * --> putlist(a(1))
            LR      R2,H
    *               going to put H + struct tag into An
            O       R2,STITAG             * set tag bits
            ST      R2,ARGS+4*1           * store value
            MVC     0(4,H),CDOTPTR        * push struct ptr
            S       H,FOUR                * and bump heap
    * --> buildvalue(p(1))
            ICM     R1,X'F',CE$VARS+4*1(CE)  * Get a perm reg ptr
    H40     BM      H41                   * No -- we are done
            C       R1,0(,R1)             * same ?
            BE      H41                   * yes -- we are done
            ICM     R1,X'F',0(R1)         * else, go another level
            BM      H41                   * No -- we are done
            C       R1,0(,R1)             * same ?
            BE      H41                   * yes -- we are done
            B       H40
            DS      0D
    H41     EQU     *
            ST      R1,0(,H)              * and push value
            S       H,FOUR                * bump heap
    * --> buildnil
            MVC     0(4,H),NILPTR         * and put constant in heap
            S       H,FOUR                * and bump heap
    * --> putvalue(a(2),p(2))
            MVC     ARGS+4*2(4),CE$VARS+4*2(CE)       * Do a MVC for speed
    * --> deallocate
            L       CP,CE$CP(,CE)         * load CP = CP(CE)
            C       CE,B$CE(,B)           * Do compare
            BNL     H42
            LR      R1,CE                 * get CE
            S       R1,FOUR                       * CE=1
            L       TOPE,0(,R1)
    *               CE(B) was < CE so TOPE = *(CE+1)
    H42     L       CE,CE$CE(,CE)         * R1 = CE(CE)
    * --> execute($LABEL(L1))
            L       BASE,=A(L1)           * load go to point
            BR      BASE                  * branch to where we should go
            DS      0D
            LTORG   Here's a nice place
            DS      0D
    * ----------> L9
    L9      EQU     *
    * --> trust_me_else_fail
            LM      B,HB,B$B(B)           * Del. bcktrck pt & get old HB
            ST      B,CUT                 * CUT = B
    * ----------> L7
    L7      EQU     *
            BALR    BASE,0
```

```
         USING   *,BASE
* --> getnils_proceed([a(0),a(1)])
         L       R2,NILPTR               * get address of nil w/ tag
         ICM     R1,X'F',ARGS+4*0        * Get an arg-reg ptr
H45      BM      H46                     * No -- we are done
         C       R1,0(,R1)                * same ?
         BE      H43                     * yes -- we are done
         ICM     R1,X'F',0(R1)           * else, go another level
         BM      H46                     * No -- we are done
         C       R1,0(,R1)                * same ?
         BE      H43                     * yes -- we are done
         B       H45
         DS      0D
H46      EQU     *
         CR      R1,R2                   *   = ?
         BE      H44
         B       FAIL                    * go fail
         DS      0D
         LTORG   Here's a nice place
         DS      0D
H43      ST      R2,0(,R1)               * at the derefed location
         CR      R1,HB                   * Lower than HB?
         BL      H48                     * Yes, no need
H47      ST      R1,0(,TR)               * save it
         S       TR,FOUR                 * push it on the stack
H48      EQU     *
H44      EQU     *
         ICM     R1,X'F',ARGS+4*1  * Get an arg-reg ptr
H51      BM      H52                     * No -- we are done
         C       R1,0(,R1)               * same ?
         BE      H50                     * yes -- we are done
         ICM     R1,X'F',0(R1)           * else, go another level
         BM      H52                     * No -- we are done
         C       R1,0(,R1)               * same ?
         BE      H50                     * yes -- we are done
         B       H51
         DS      0D
H52      EQU     *
         CR      R1,R2                   *   = ?
         BER     CP
         B       FAIL                    * go fail
         DS      0D
         LTORG   Here's a nice place
         DS      0D
H50      ST      R2,0(,R1)               * at the derefed location
         CR      R1,HB                   * Lower than HB?
         BLR     CP                      * Yes, no need
H53      ST      R1,0(,TR)               * save it
         S       TR,FOUR                 * push it on the stack
         BR      CP
```

G Future IBM-370 Assembly

Of the optimizations discussed in this paper, six have not yet been implemented. These are:

- Unbound variables represented by zeros and the reservation of a general purpose register to contain zero.
- After doing the above, restore the "unsafe" instructions.
- Heap and Code space growing in the opposite direction from the other stacks.
- Reserving a register to contain a pointer to the structure representing a list (CDOTPTR).

- Certain timing optimizations.
- The trailing optimization for determinate procedures (like append).

The first of these optimizations reduces stack setup time in allocate as well as making dereferencing time shorter. The second reduces arithmetic instructions by replacing S (subtract) instructions with LA instruction (which do not reference memory, or take much time to execute).

Eventual code improvements should result in a main loop of around 22 instructions with several fewer memory accesses and speeds in excess of 1.1 MegaLips.

H Hints to other Prolog compiler writers

H.1 Why native code generation is better than byte code

If we had generated byte code instead of native code, our memory usage in the code area would be roughly a factor of eight or ten smaller. While for certain applications this may be useful, I argue that these cases are the minority. The cost of memory is falling at a much more rapid rate than machines are getting faster. In addition, when processing needs increase, it is much easier to get new memory than it is to get an entirely new processor.

H.2 Commented code generator

Our code generator produced IBM assembly code, written to a file. This code was then run through the IBM assembler. By including in the generated code many comments, it was much easier to remember why a particular sequence of code was used. Looking in the appendices, one can see the style of comments that we used.

After the compiler has been finished, it is easy to modify emit so that comments are no longer generated in the code generator. Ideally native binary code would be (eventually) produced, so that an in-core compiler would not be too slow.

H.3 WAM Instruction and Register Traces

One of the most useful tools in debugging the compiler was an option of the code generator that produced a small macro at the beginning of each WAM instruction. This macro called a small debug package that checked a user settable flag. If the flag was on, a WAM instruction trace and / or a WAM register trace was printed.

This trace often saved us many hours of debugging time by allowing us to quickly find out where the compiler was going to. In addition, it provides useful performance statistics – by writing out just the WAM instructions and then sorting and counting these, very accurate statistics could be gathered.

References

[1] Hughes, K., *An Implementation of Prolog,* Caltech Computer Science Internal Memo, 1985.

[2] IBM Corporation, *IBM System/370 Principles of Operation* GA22-7004-4, Poughkeepsie, NY, May 1983.

[3] IBM Corporation, *IBM System/370 Extended Architecture Principles of Operation* SA22-7085-0, Poughkeepsie, NY, March 1983 and November 1985.

[4] Overbeek, R., *A Short Note on If-Then-Else, Prolog Style, and Efficient Compilation to The Warren Abstract Machine* Argonne National Lab.s, Argonne Illinois.

[5] Pereira, F., Warren, D., Bowen, D., Byrd, L., and Pereira, L., *C-Prolog Users Manual* SRI International.

[6] Tick, E., *Prolog Memory-Referencing Behavior,* Technical Report No. 85-281, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, September 1985.

[7] Turk, A. K., *Compiler Optimizations for the WAM,* School of Computer and Information Science, Syracuse University, Technical Report CIS-85-6, November 1985.

[8] Van Roy, P., *A Prolog Compiler for the PLM,* Master's Report, Computer Science Division, University of California, August 21, 1984.

[9] Warren, David H. D., *Implementing Prolog – Compiling Predicate Logic Programs,* University of Edinburgh Department of Artificial Intelligence Report 39, May 1977.

[10] Warren, David H. D., *Implementing Prolog – Compiling Predicate Logic Programs,* University of Edinburgh Department of Artificial Intelligence Report 40, May 1977.

[11] Warren, David H. D., *An Improved Prolog Implementation which Optimizes Tail Recursion,* University of Edinburgh Department of Artificial Intelligence Research Paper 156, 1980, (Also presented at the 1980 Logic Programming Workshop, Debrecen, Hungary).

[12] Warren, David H. D., *An Abstract Prolog Instruction Set,* Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Technical Note 309, October 1983.